First Python Notebook

unknown

Jul 17, 2023

CHAPTERS

A step-by-step guide to analyzing data with Python and the Jupyter notebook.

CHAPTER

ONE

WHAT YOU WILL LEARN

This textbook will teach you:

- Just enough of the Python computer-programming language to read, filter, join, group, aggregate and rank structured data with pandas, a popular open-source tool for statistical analysis
- How to record, remix and republish your work using Project Jupyter, a browser-based interface for writing code that is emerging as the standard for generating reproducible research
- How to explore data using using Altair, a Python package that offers a simple, structured grammar for generating charts.

CHAPTER

TWO

WHO CAN TAKE IT

This course is free. If you've tried Python once or twice, have good attitude and know how to take a few code crashes in stride, you are qualified.

CHAPTER

THREE

TABLE OF CONTENTS

3.1 JupyterLab

A Jupyter notebook is a browser-based interface where you can write, run, remix and republish code.

It is free software you can install and run like any other open-source library. It is used by scientists, scholars, investors and corporations to create and share their research.

It is also used by journalists to develop stories and show their work. Examples published by past teachers of this class include:

- "As Opioid Crisis Ramped Up, Pills Flowed Into Vermont by the Millions" by Andrea Suozzo
- "A frenzy of well drilling is depleting aquifers in California farmland." by Gabrielle LaMarr LeMee
- "What it's like to go to school when dozens have been killed nearby" by Iris Lee
- "City of Chicago Parking and Camera Ticket Data" by David Eads

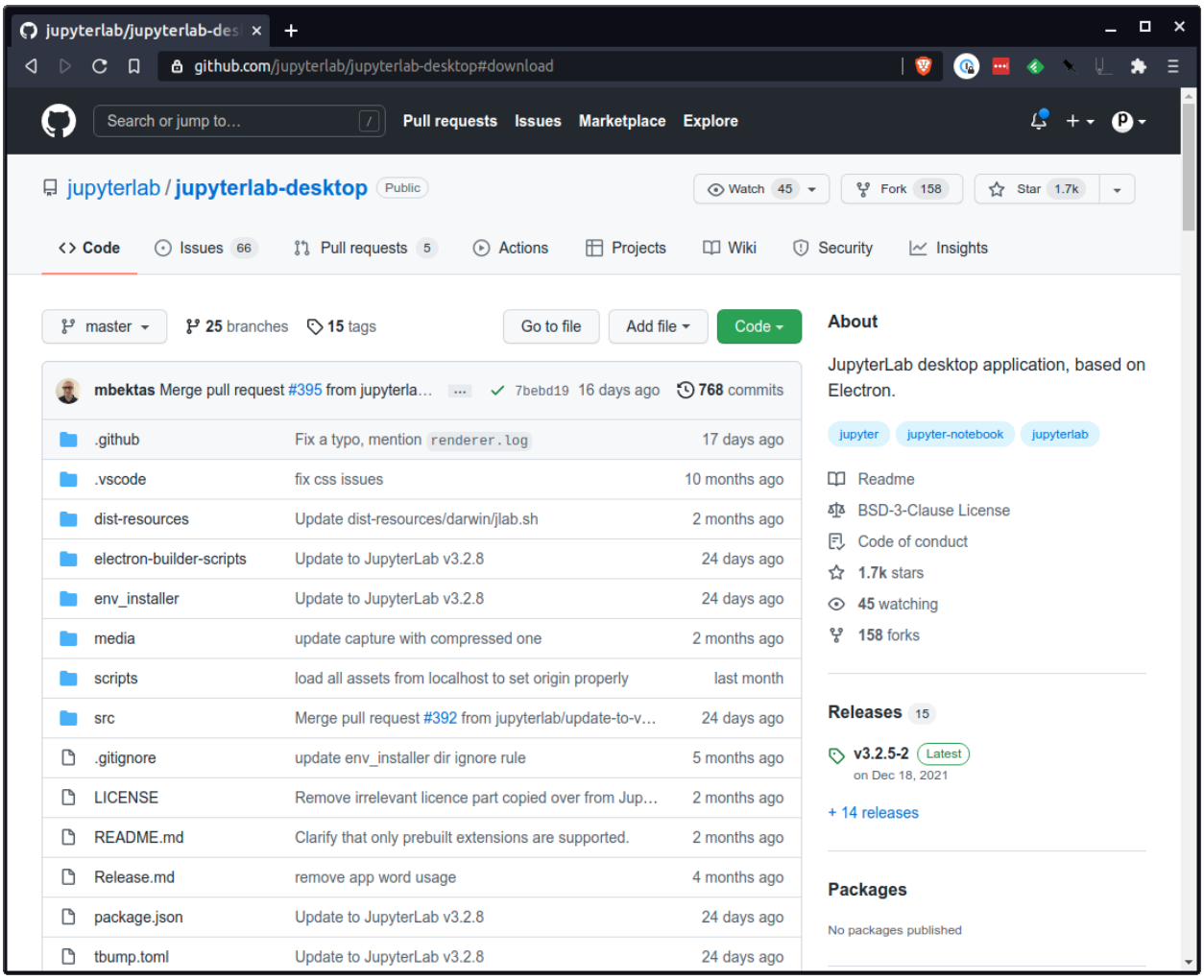
You can find hundreds of other examples on GitHub, including work by Buzzfeed, ProPublica, The Economist, POLITICO, The Markup and the Los Angeles Times.

There are numerous ways to install and configure Jupyter notebooks. Since this tutorial is designed for beginners, it will demonstrate how to use JupyterLab Desktop, a self-contained application that provides a ready-to-use Python environment with several popular libraries bundled in. It can be installed on any operating system with a simple point-and-click interface.

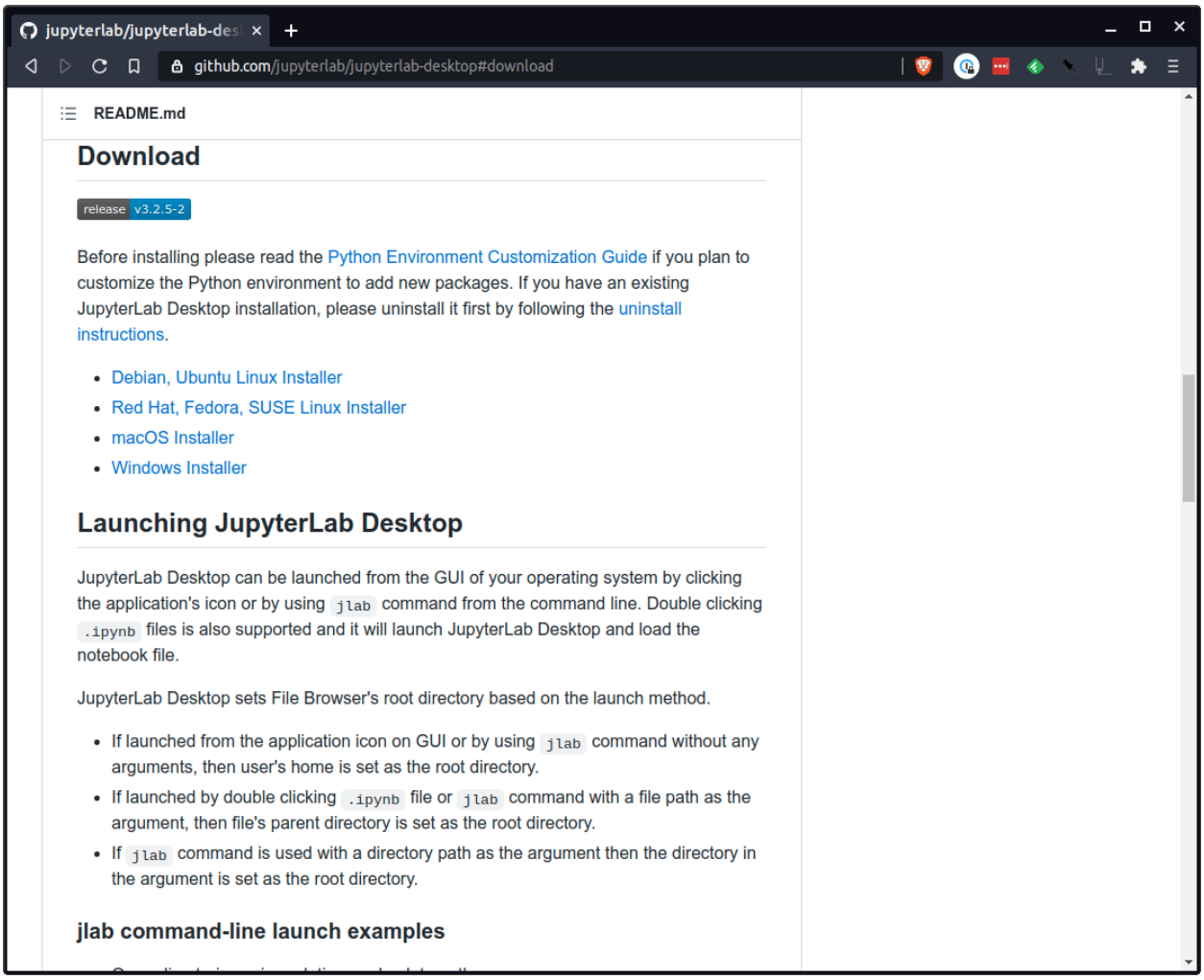
Note: Advanced users like to take advantage of Python's power tools to have more control over when and where code is installed on their system. Readers interested in the techniques preferred by the pros should consult *our appendix*. It requires use of your computer's command-line interface.

3.1.1 Install JupyterLab Desktop

The first step is to visit JupyterLab Desktop's homepage on GitHub in your web browser.



Scroll down to the documentation below the code until you reach the Download section.



Then pick the link appropriate for your operating system. The installation file is large so the download might take a while.

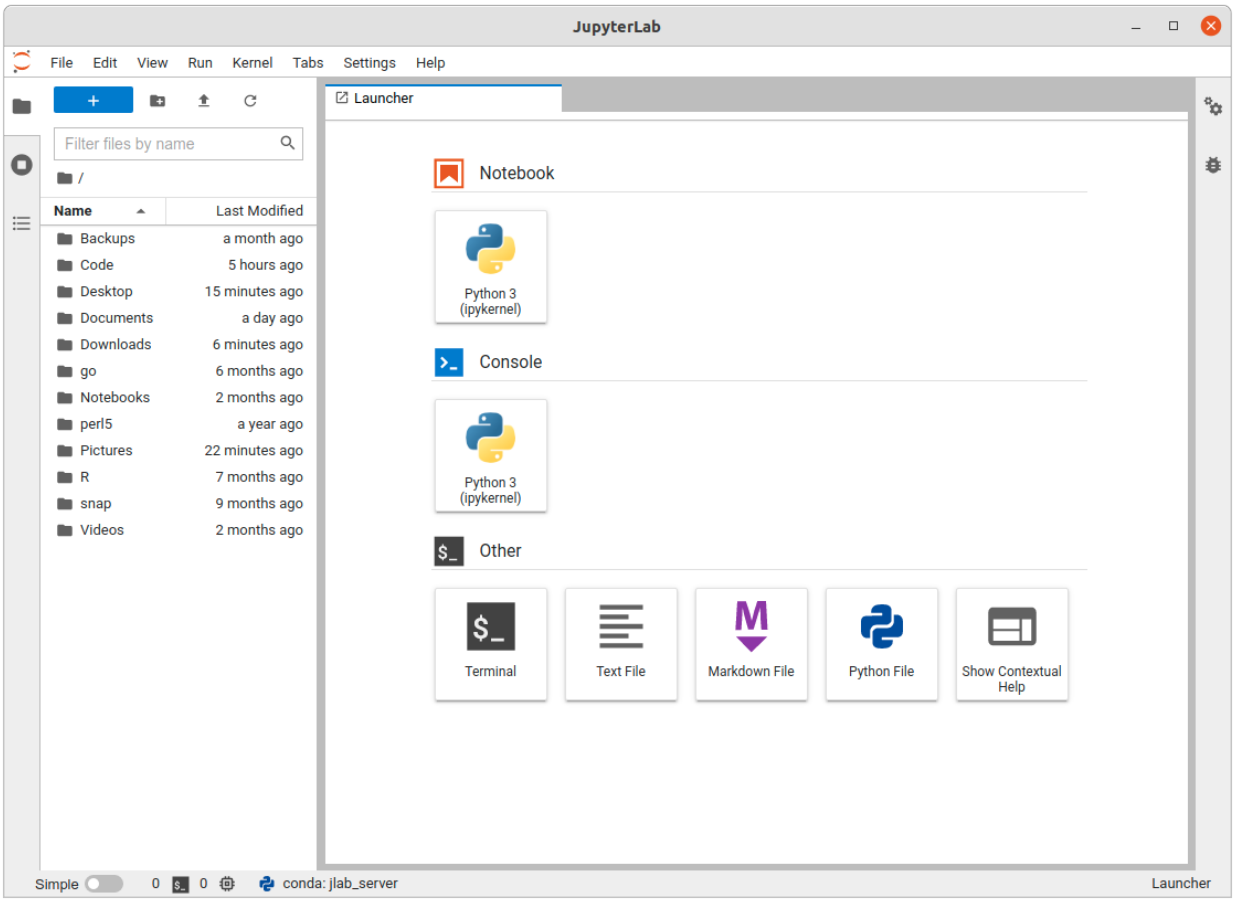
Find the file in your downloads directory and double click it to begin the installation process. Follow the instructions presented by the pop-up windows, sticking to the default options.

> **Warning:** Your computer's operating system might flag the JupyterLab Desktop installer as an unverified or insecure application. Don't worry. The tool has been vetted by Project Jupyter's core developers and it's safe to use.
>
> If your system is blocking you from installing the tool, you'll likely need to work around its barriers. For instance, on MacOS, this might require visiting your system's security settings to allow the installation.

3.1.2 Open a Python 3 notebook

Once the program is installed, you can accept the installation wizard's offer to immediately open the program, or you can search for "Jupyter Lab" in your operating system's application finder.

That will open up a new window that looks something like this:



Hit the "Python 3" button in the launcher panel on the right and you're ready to move on to our next chapter.

3.2 Notebooks

You should see a new panel with an empty box at the top. That means you are all set up and ready to write Python. If you've never done it before, you can remain calm. We can start out slow with some simple math.

Type the following into the box, then hit the play button in the toolbar above the notebook or hit `SHIFT+ENTER` on your keyboard. The number four should appear.

```
2+2
```

```
4
```

There. Not so bad, right? You have just written your first code. When you execute a cell, the text inside it will be processed and the output will be displayed below the cell. If the cell contains code, it will be run by the Jupyter

notebook's underlying programming language. In the jargon of Python, you have entered two integers and combined them using the addition operator.

Now try writing in your own math problem in the next cell. Maybe `2+3` or `2+200`. Whatever strikes your fancy. After you've typed it in, hit the play button or `SHIFT+ENTER`. This to-and-fro of writing Python code in a cell and then running it with the play button is the rhythm of working in a notebook.

If you get an error after you run a cell, look carefully at your code and see that it exactly matches what's been written in the example. Here's an example of a error that I've added intentionally:

```
2+2+
```

```
  File "/tmp/ipykernel_952/4150814810.py", line 1
    2+2+
        ^
SyntaxError: invalid syntax
```

Don't worry. Code crashes are a normal part of life for computer programmers. They're usually caused by small typos that can be quickly corrected.

```
2+2+2
```

```
6
```

The best thing you can do is remain calm and carefully read the error message. It usually contains clues that can help you fix the problem.

Over time you will gradually stack cells to organize an analysis that runs down your notebook from the top. A simple example is storing your number in a variable in one cell:

```
number = 2
```

Then adding it to another value in the next cell:

```
number + 3
```

```
5
```

Run those two cells in succession and the notebook should output the number five.

Change the `number` value to 3 and run both cells again. Instead of 5, it should now output 6.

So, first this:

```
number = 3
```

Then this:

```
number + 3
```

```
6
```

Now try defining your own numeric variable and doing some math with it. You can name it whatever you want. Want to try some other math operations? The `-` sign does subtraction. Multipication is `*`. Division is `/`.

Sometimes it is helpful to describe what the code is doing in case you want to share it with a colleague or return to it after some time. You can add comments in the cell by putting a hash `#` in front of the text. So, for example, we could use a comment to add extra information about the number variable.

```
# This is a random number
number = 3
```

To add a cell in a Jupyter notebook, you can use the "Insert" menu at the top of the page and select "Insert Cell Above" or "Insert Cell Below". Alternatively, you can use the keyboard shortcut "a" to insert a cell above the current cell or "b" to insert a cell below the current cell. You can also use the "+" button in the toolbar above the notebook to insert a cell below the current cell.

To remove a cell, you can select the cell and press the "dd" key. Alternatively, you can use the "Edit" menu at the top of the page and select "Delete Cells" or you can use the "scissors" button in the toolbar above the notebook to delete the selected cell. Note that when you delete a cell, everything in that cell will be lost and it cannot be undone.

Note: Cells can contain variables, functions or imports. If you've never written code before and are unfamiliar with those terms, we recommend "An Informal Introduction to Python" and subsequent sections of python.org's official tutorial.

Everything we have done so far has been in code cells, the default cell type. We can also make text cells, which are useful for giving our notebooks some structure and organization. You can do this by manipulating the pulldown menu in the toolbar directly above the notebook. By default the input is set to "Code." Click the dropdown arrow and change it to "Markdown," a markup language for formatting text similar to HTML.

These cells allow you to create headers, write descriptions, add links and more to add context to your code. "The Ultimate Markdown Guide" is a reference on all of the styling that you can draw from. For now, let's try adding a heading and a bulleted list.

```
## Hashes make headings
- Dashes make
- Bulleted lists
```

3.2.1 Hashes make headings

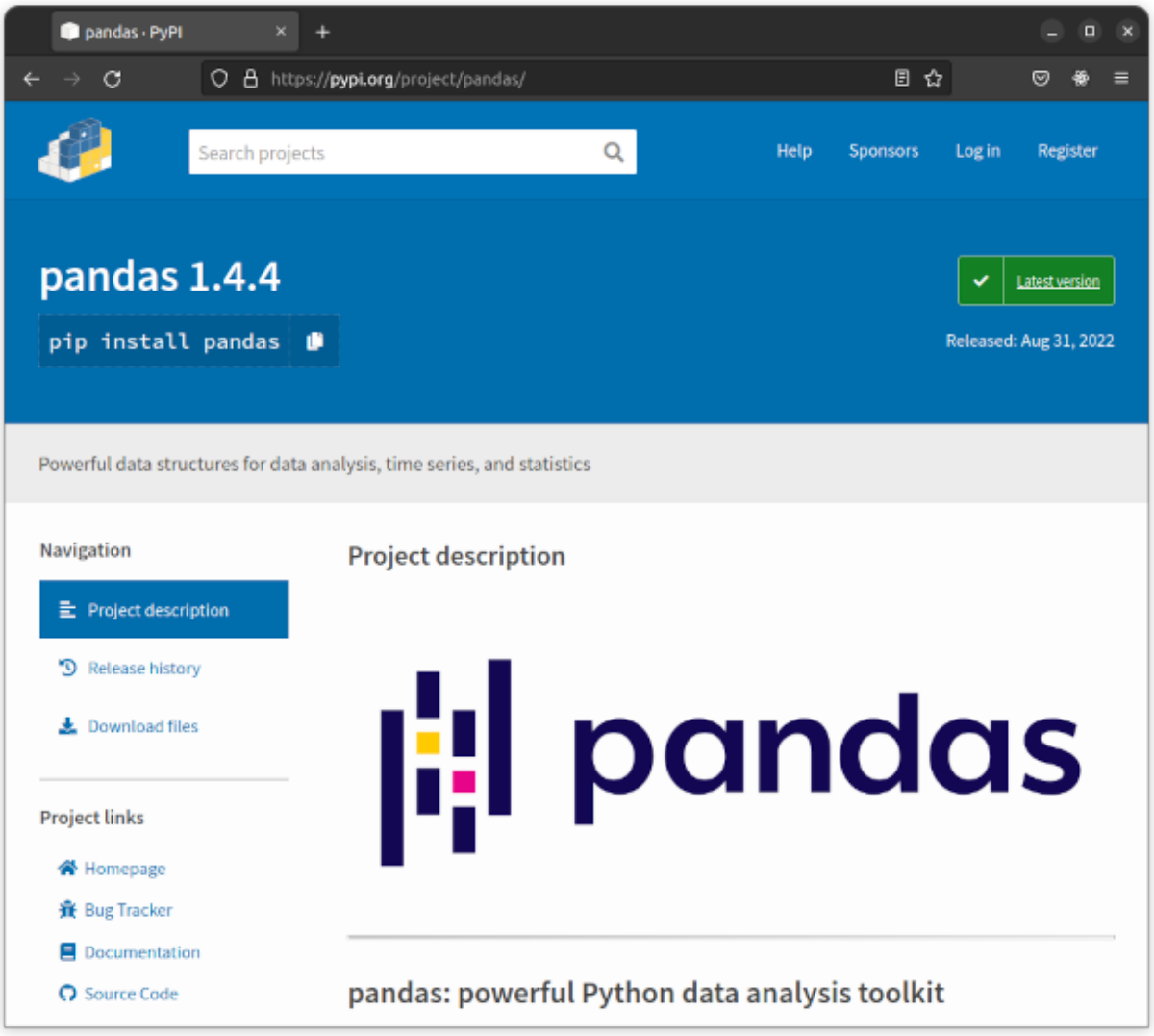
- Dashes make
- Bulleted lists

Once you've got the hang of making the notebook run, you're ready to introduce pandas, the powerful Python analysis library that can do a whole lot more than add a few numbers together.

3.3 Pandas

Python is filled with functions to do pretty much anything you'd ever want to do with a programming language: navigate the web, parse data, interact with a database, run fancy statistics, build a pretty website and so much more.

Creative people have put these tools to work to get a wide range of things done in the academy, the laboratory and even in outer space. Some are included in a toolbox that comes with the language, known as the standard library. Others have been built by members of Python's developer community and need to be downloaded and installed from the web.



One third-party tool that's important for this class is called pandas. It was invented for use at a financial investment firm and has become the leading open-source library for accessing and analyzing data in many different fields.

3.3.1 Import pandas

Create a new cell at the top of your notebook where we will import pandas for our use. Type in the following and hit the play button.

```
import pandas
```

If nothing happens, that's good. It means you have pandas installed and ready as to use.

Note: Since pandas is created by a third party independent from the core Python developers, it wouldn't be installed by default if you followed our *our advanced installation* instructions.

It's available to you because the JupyterLab Desktop developers have pre-selected a curated list of common utilities to include with the package, another reason to love their easy installer.

Return to the cell with the import and rewrite it like this.

```
import pandas as pd
```

This will import the pandas library at the shorter variable name of `pd`. This is standard practice in the pandas community. You will frequently see examples of pandas code online using `pd` as shorthand. It's not required, but it's good to get in the habit so that your code is more likely to be quickly understood by other computer programmers.

Note: In Python, a variable is a way to store a value in memory for later use. A variable is a named location in the computer's memory where a value can be stored and retrieved. Variables are used to store data values, such as numbers, strings, lists, or objects, and they can be used throughout the program to refer to the stored value.

To create your own variable in Python, you use the assignment operator (=) to assign a value to a variable. The variable name is on the left side of the assignment operator and the value is on the right side.

3.3.2 Conduct a simple data analysis

Those two little letters contain dozens of data analysis tools that we'll use in future lessons. They can read in millions of records, compute advanced statistics, filter, sort, rank and do just about anything else you'd want to do with data.

We'll get to all of that soon enough, but let's start out with something simple.

Let's make a list of numbers in a new notebook cell. To keep things simple, enter all of the even numbers between zero and ten. Name its variable something plain like `my_list`. Press play.

```
my_list = [2, 4, 6, 8]
```

You can do cool stuff with any list, even calculate advanced statistics, if you're a skilled Python programmer who is ready and willing to write a big chunk of code. The advantage of pandas is that it saves time by quickly and easily analyzing data with hardly any computer code at all.

In this case, it's as simple as converting that plain Python list into what pandas calls a Series. Here's how to make it happen in your next cell. Let's stick with simple variables and name it `my_series`.

```
my_series = pd.Series(my_list)
```

Once the data becomes a Series, you can immediately run a wide range of descriptive statistics. Let's try a few.

How about summing all the numbers? Make a new cell and run this. It should spit out the total.

```
my_series.sum()
```

```
20
```

Then find the maximum value in the next.

```
my_series.max()
```

```
8
```

The minimum value in the next.

```
my_series.min()
```

```
2
```

How about the average, which also known as the mean?

```
my_series.mean()
```

```
5.0
```

The median?

```
my_series.median()
```

```
5.0
```

The standard deviation?

```
my_series.std()
```

```
2.581988897471611
```

Finally, all of the above, plus a little more about the distribution, in one simple command.

```
my_series.describe()
```

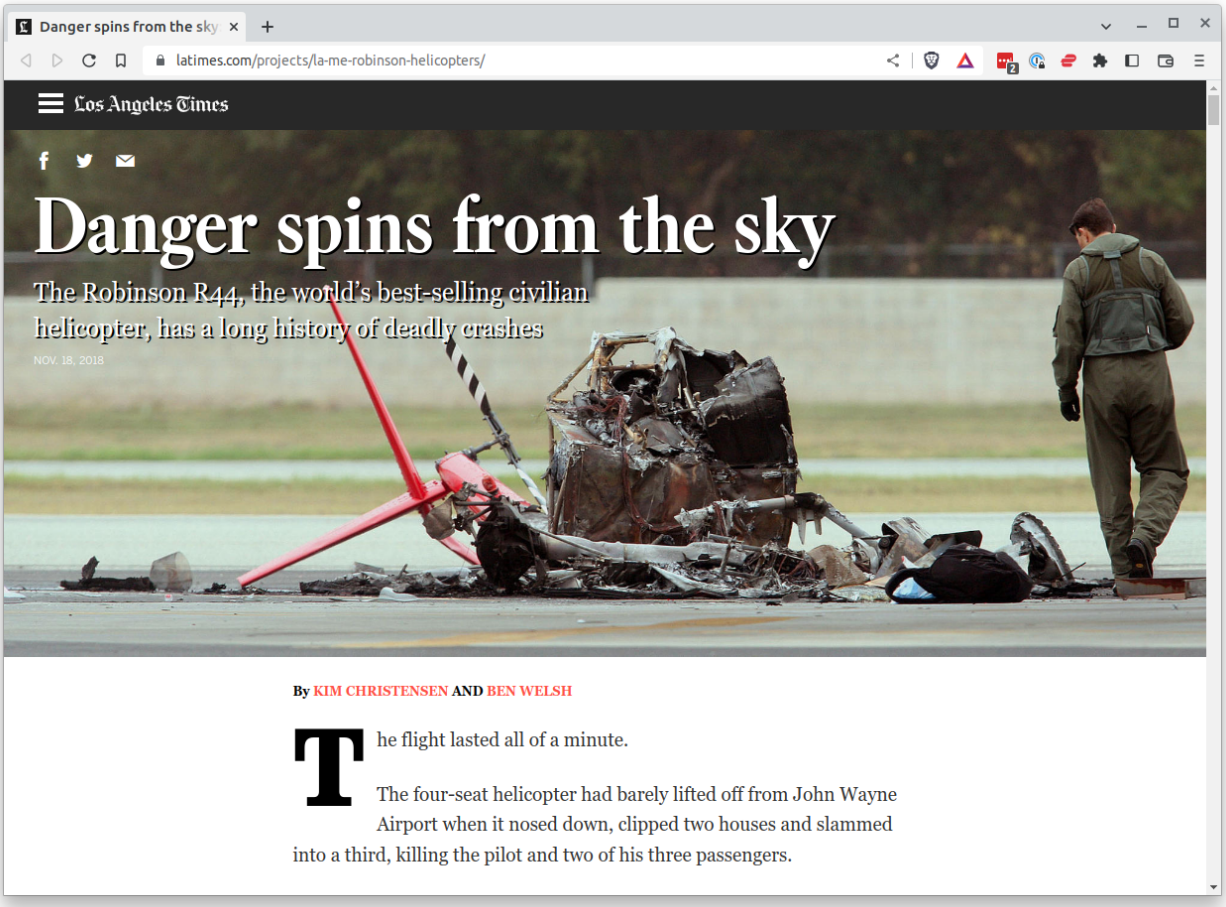
```
count    4.000000
mean     5.000000
std      2.581989
min      2.000000
25%      3.500000
50%      5.000000
75%      6.500000
max      8.000000
dtype: float64
```

Before you move on, go back to the cell with your `my_list` variable and change what's in the list. Maybe add a few more values. Or switch from evens to odds. Then rerun all the cells below it. You'll see all the statistics update to reflect the different dataset.

If you substituted in a series of 10 million records, your notebook would calculate all those same statistics without you needing to write any more code. Once your data, however large or complex, is imported into pandas, there's little limit to what you can do to filter, merge, group, aggregate, compute or chart using simple methods like the ones above. In the chapter to come we'll start doing just using that with data from a real Los Angeles Times investigation.

3.4 Data

In 2018, the Los Angeles Times published an investigation headlined, "The Robinson R44, the world's best-selling civilian helicopter, has a long history of deadly crashes."



It reported that Robinson's R44 led all major models with the highest fatal accident rate from 2006 to 2016. The analysis was published on GitHub as a series of Jupyter notebooks.

The findings were drawn from two key datasets:

1. The National Transportation Safety Board's Aviation Accident Database
2. The Federal Aviation Administration's General Aviation and Part 135 Activity Survey

After a significant amount of work gathering and cleaning the source data, the number of accidents for each helicopter model were normalized using the flight hours estimates in the survey. For the purposes of this demonstration, we will read in tidied versions of each file that are ready for analysis.

The data are structured in rows of comma-separated values. This is known as a CSV file. It is the most common way you will find data published online. The pandas library is able to read in files from a variety formats, including CSV.

```
import pandas as pd
```

3.4.1 The `read_csv` method

Scroll down to the first open cell. There we will import the first CSV file using the `read_csv` function included with pandas.

```
pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-notebook/main/docs/
↪src/_static/ntsb-accidents.csv")
```

```
           event_id                        ntsb_make ntsb_model ntsb_number  \
0    20061222X01838                             BELL        407  NYC07FA048
1    20060817X01187                         ROBINSON   R22 BETA  LAX06LA257
2    20060111X00044                         ROBINSON        R44  MIA06FA039
3    20060419X00461                         ROBINSON     R44 II  DFW06FA102
4    20060208X00181                         ROBINSON        R44  SEA06LA052
..              ...                              ...        ...         ...
158  20160711X32921  BELL HELICOPTER TEXTRON CANADA        407  ERA16FA248
159  20160804X45514                       SCHWEIZER     269C 1  CEN16FA304
160  20160404X74644                            BELL        206  ERA16FA144
161  20160507X31120                          AIRBUS      AS350  ANC16FA023
162  20160612X85856     ROBINSON HELICOPTER COMPANY     R44 II  CEN16FA215

     year               date            city state country  total_fatalities  \
0    2006  12/14/06 00:00:00        DAGSBORO    DE     USA                 2
1    2006  08/10/06 00:00:00          TUCSON    AZ     USA                 1
2    2006  01/01/06 00:00:00     GRAND RIDGE    FL     USA                 3
3    2006  04/13/06 00:00:00  FREDERICKSBURG    TX     USA                 2
4    2006  02/06/06 00:00:00          HELENA    MT     USA                 1
..    ...                ...             ...   ...     ...               ...
158  2016  07/11/16 00:00:00         HICKORY    KY     USA                 1
159  2016  08/03/16 00:00:00      JEANERETTE    LA     USA                 1
160  2016  04/04/16 00:00:00    PIGEON FORGE    TN     USA                 5
161  2016  05/06/16 00:00:00          SKAGWAY    AK     USA                 1
162  2016  06/12/16 00:00:00        JONESBORO    AR     USA                 1

    latimes_make latimes_model latimes_make_and_model
0           BELL           407               BELL 407
1       ROBINSON           R22           ROBINSON R22
2       ROBINSON           R44           ROBINSON R44
3       ROBINSON           R44           ROBINSON R44
4       ROBINSON           R44           ROBINSON R44
..           ...           ...                   ...
158         BELL           407               BELL 407
159    SCHWEIZER           269          SCHWEIZER 269
160         BELL           206               BELL 206
161       AIRBUS           350             AIRBUS 350
162     ROBINSON           R44           ROBINSON R44

[163 rows x 13 columns]
```

Warning: You will need to precisely type in the URL to the file. Feel free to copy and paste it from the example above into your notebook.

After you run the cell, you should see a big table output to your notebook. It is a "DataFrame" where pandas has structured the CSV data into rows and columns, just like Excel or other spreadsheet software might. Take a moment to look at the columns and rows in the output, which contain the data we'll use in our analysis.

Note: On the left-hand side, you'll see an bolded number incrementing up from zero that present in our source data file. This what pandas calls the index. It is a separate column created automatically that is used to identify each row. The index is not considered part of the data, but is used to reference the rows of the DataFrame or Series in advanced operations that are beyond the scope of this class.

A major advantage of Jupyter over spreadsheets is that rather than manipulating the data through a haphazard series of clicks and keypunches we will be gradually grinding it down using a computer programming script that is transparent and reproducible.

In order to do more with your DataFrame, we need to store it so it can be reused in subsequent cells. We can do this by saving in a variable, just as we did in with our `number` in chapter two.

Go back to your latest cell and change it to this. Rerun it.

```
accident_list = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-
↪notebook/main/docs/src/_static/ntsb-accidents.csv")
```

You shouldn't see anything. That's a good thing. It means our DataFrame has been saved under the name `accident_list`, which we can now begin interacting with in the cells that follow.

We can do this by calling "methods" that pandas makes available to all DataFrames. You may not have known it at the time, but `read_csv` is one of these methods. There are dozens more that can do all sorts of interesting things. Let's start with some easy ones that analysts use all the time.

3.4.2 The `head` method

To preview the first few rows of the dataset, try the `head` method. Add a new cell and type this in and hit the play button again.

```
accident_list.head()
```

```
         event_id ntsb_make ntsb_model ntsb_number  year               date  \
0  20061222X01838      BELL         407  NYC07FA048  2006  12/14/06 00:00:00
1  20060817X01187  ROBINSON   R22 BETA  LAX06LA257  2006  08/10/06 00:00:00
2  20060111X00044  ROBINSON         R44  MIA06FA039  2006  01/01/06 00:00:00
3  20060419X00461  ROBINSON      R44 II  DFW06FA102  2006  04/13/06 00:00:00
4  20060208X00181  ROBINSON         R44  SEA06LA052  2006  02/06/06 00:00:00

             city state country  total_fatalities latimes_make latimes_model  \
0        DAGSBORO    DE     USA                 2         BELL           407
1          TUCSON    AZ     USA                 1     ROBINSON           R22
2     GRAND RIDGE    FL     USA                 3     ROBINSON           R44
3  FREDERICKSBURG    TX     USA                 2     ROBINSON           R44
4          HELENA    MT     USA                 1     ROBINSON           R44

  latimes_make_and_model
0               BELL 407
1           ROBINSON R22
2           ROBINSON R44
```

(continues on next page)

(continued from previous page)

```
3           ROBINSON R44
4           ROBINSON R44
```

It does the first five by default. If you want a different number, submit it as an input.

```
accident_list.head(1)
```

```
         event_id ntsb_make ntsb_model ntsb_number  year               date  \
0  20061222X01838      BELL        407  NYC07FA048  2006  12/14/06 00:00:00

       city state country  total_fatalities latimes_make latimes_model  \
0  DAGSBORO    DE     USA                 2         BELL           407

  latimes_make_and_model
0               BELL 407
```

3.4.3 The `info` method

To get a look at all of the columns and what type of data they store, add another cell and try the info method. Look carefully at the results and you'll see we have 163 fatal accidents to review.

```
accident_list.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 163 entries, 0 to 162
Data columns (total 13 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   event_id               163 non-null    object
 1   ntsb_make              163 non-null    object
 2   ntsb_model             163 non-null    object
 3   ntsb_number            163 non-null    object
 4   year                   163 non-null    int64
 5   date                   163 non-null    object
 6   city                   163 non-null    object
 7   state                  162 non-null    object
 8   country                163 non-null    object
 9   total_fatalities       163 non-null    int64
 10  latimes_make           163 non-null    object
 11  latimes_model          163 non-null    object
 12  latimes_make_and_model  163 non-null    object
dtypes: int64(2), object(11)
memory usage: 16.7+ KB
```

Now that you've got some data imported, we're ready to begin our analysis.

3.5 Columns

We'll begin with the `latimes_make_and_model` column, which records the standardized name of each helicopter that crashed. To access its contents separate from the rest of the DataFrame, append a pair of flat brackets with the column's name in quotes inside.

```
import pandas as pd
accident_list = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-notebook/main/docs/src/_static/ntsb-accidents.csv")
```

```
accident_list['latimes_make_and_model']
```

```
0           BELL 407
1       ROBINSON R22
2       ROBINSON R44
3       ROBINSON R44
4       ROBINSON R44
           ...
158         BELL 407
159    SCHWEIZER 269
160         BELL 206
161       AIRBUS 350
162     ROBINSON R44
Name: latimes_make_and_model, Length: 163, dtype: object
```

That will list the column out as a `Series`, just like the ones we created from scratch earlier. Just as we did then, you can now start tacking on additional methods that will analyze the contents of the column.

Note: You can also access columns a second way, like this: `accident_list.latimes_make_and_model`. This method is quicker to type, but it won't work if your column has a space in its name. So we're teaching the universal bracket method instead.

3.5.1 Count a column's values

In this case, the column is filled with characters. So we don't want to calculate statistics like the median and average, as we did before.

There's another built-in pandas tool that will total up the frequency of values in a column. The method is called `value_counts` and it's just as easy to use as `sum`, `min` or `max`. All you need to do it is add a period after the column name and chain it on the tail end of your cell.

```
accident_list['latimes_make_and_model'].value_counts()
```

```
ROBINSON R44             38
BELL 206                 30
AIRBUS 350               29
ROBINSON R22             20
BELL 407                 13
HUGHES 369               13
MCDONNELL DOUGLAS 369     6
```

(continues on next page)

(continued from previous page)

```
SCHWEIZER 269              5
AIRBUS 135                 4
SIKORSKY 76                2
AGUSTA 109                 2
AIRBUS 130                 1
Name: latimes_make_and_model, dtype: int64
```

Congratulations, you've made your first finding. With that little line of code, you've calculated an important fact: During the period being studied, the Robinson R44 had more fatal accidents than any other helicopter.

3.5.2 Reset a DataFrame

You may notice that even though the result has two columns, pandas did not return a clean-looking table in the same way as `head` did for our DataFrame. That's because our column, a Series, acts a little bit different than the DataFrame created by `read_csv`. In most instances, you can convert ugly Series into a pretty DataFrame by tacking on the `reset_index` method on the end.

```
accident_list['latimes_make_and_model'].value_counts().reset_index()
```

	index	latimes_make_and_model
0	ROBINSON R44	38
1	BELL 206	30
2	AIRBUS 350	29
3	ROBINSON R22	20
4	BELL 407	13
5	HUGHES 369	13
6	MCDONNELL DOUGLAS 369	6
7	SCHWEIZER 269	5
8	AIRBUS 135	4
9	SIKORSKY 76	2
10	AGUSTA 109	2
11	AIRBUS 130	1

Why does a Series behave differently than a DataFrame? Why does `reset_index` have such a weird name?

Like so much in computer programming, the answer is simply, "because the people who created the library said so." It's important to learn that all open-source programming tools are made by humans, and humans have their quirks. Over time you'll see pandas has more than a few.

As a beginner, you should just accept the oddities and keep moving. As you get more advanced, if there's something about the system you think could be improved you should consider contributing to the Python code that operates the library.

Before we move on to the next chapter, here's a challenge. See if you can answer a few more questions a journalist might ask about our dataset. All four of the questions below can be answered using only tricks we've covered thus far. See if you can do it.

1. What was the total number of fatalities?
2. Which helicopter maker had the most accidents?
3. What was the total number of helicopter accidents by year?
4. Which state had the most helicopter accidents?

Once you've written code answered those, you're ready to move on to the next chapter.

3.6 Filter

The most common way to filter a DataFrame is to pass an expression as an "index" that can be used to decide which records should be kept and which discarded. You write the expression by combining a column on your DataFrame with an "operator" like == or > or < and a value to compare against each row.

Note: If you are familiar with writing SQL to manipulate databases, pandas' filtering system is somewhat similar to a WHERE query. The official pandas documentation offers direct translations between the two.

Let's try filtering against the `state` field. Save a postal code into a variable. This will allow us to reuse it later.

```
import pandas as pd
accident_list = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-
↪notebook/main/docs/src/_static/ntsb-accidents.csv")
```

```
my_state = "IA"
```

In the next cell we will ask pandas to narrow down our list of accidents to just those in the state we're interested in. We will create a filter expression and place it between two flat brackets following the DataFrame we wish to filter.

```
accident_list[accident_list['state'] == my_state]
```

```
          event_id                    ntsb_make ntsb_model ntsb_number  year  \
12   20060705X00868                         BELL       206B  CHI06FA173  2006
118  20130102X35708               BELL HELICOPTER        407  CEN13FA122  2013
140  20140908X10448  ROBINSON HELICOPTER COMPANY     R44 II  CEN14LA487  2014

                  date        city state country  total_fatalities  \
12   06/30/06 00:00:00     WALFORD    IA     USA                 1
118  01/02/13 00:00:00  CLEAR LAKE    IA     USA                 3
140  09/06/14 00:00:00   MACEDONIA    IA     USA                 1

    latimes_make latimes_model latimes_make_and_model
12          BELL           206               BELL 206
118         BELL           407               BELL 407
140     ROBINSON           R44           ROBINSON R44
```

Now we should save the results of that filter into a new variable separate from the full list we imported from the CSV file. Since it includes only the sites for the state we want, let's call it `my_accidents`.

```
my_accidents = accident_list[accident_list['state'] == my_state]
```

To check our work and find out how many records are left after the filter, let's run the DataFrame inspection commands we learned earlier.

First `head`.

```
my_accidents.head()
```

```
          event_id                    ntsb_make ntsb_model ntsb_number  year  \
12   20060705X00868                         BELL       206B  CHI06FA173  2006
118  20130102X35708               BELL HELICOPTER        407  CEN13FA122  2013
```

(continues on next page)

(continued from previous page)

```
140  20140908X10448  ROBINSON HELICOPTER COMPANY     R44 II  CEN14LA487  2014

                  date        city state country  total_fatalities  \
12   06/30/06 00:00:00     WALFORD    IA     USA                 1
118  01/02/13 00:00:00  CLEAR LAKE    IA     USA                 3
140  09/06/14 00:00:00   MACEDONIA    IA     USA                 1

    latimes_make latimes_model latimes_make_and_model
12          BELL           206               BELL 206
118         BELL           407               BELL 407
140     ROBINSON           R44            ROBINSON R44
```

Then `info`.

```
my_accidents.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3 entries, 12 to 140
Data columns (total 13 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   event_id                3 non-null      object
 1   ntsb_make               3 non-null      object
 2   ntsb_model              3 non-null      object
 3   ntsb_number             3 non-null      object
 4   year                    3 non-null      int64
 5   date                    3 non-null      object
 6   city                    3 non-null      object
 7   state                   3 non-null      object
 8   country                 3 non-null      object
 9   total_fatalities        3 non-null      int64
 10  latimes_make            3 non-null      object
 11  latimes_model           3 non-null      object
 12  latimes_make_and_model  3 non-null      object
dtypes: int64(2), object(11)
memory usage: 336.0+ bytes
```

Now pick another state and try running the code again. See if you can write filters that will answer the following questions:

1. Which state recorded more accidents, Iowa or Missouri?
2. How many accidents recorded more than one fatality?
3. How many accidents happened in California in 2015?
4. What percentage of the total fatalities occured in California?

3.7 Group

The `groupby` method allows you to group a DataFrame by a column and then calculate a sum, or any other statistic, for each unique value. This functions much like the "pivot table" feature found in most spreadsheets.

Let's use it to total up the accidents by make and model. You start by passing the field you want to group on to the function.

```
import pandas as pd
accident_list = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-notebook/main/docs/src/_static/ntsb-accidents.csv")
```

```
accident_list.groupby("latimes_make_and_model")
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7efe8feae190>
```

A nice start but you'll notice you don't get much back. The data's been grouped, but we haven't chosen what to do with it yet. If we wanted the total by model, we would use the `size` method.

```
accident_list.groupby("latimes_make_and_model").size()
```

```
latimes_make_and_model
AGUSTA 109                 2
AIRBUS 130                 1
AIRBUS 135                 4
AIRBUS 350                29
BELL 206                  30
BELL 407                  13
HUGHES 369                13
MCDONNELL DOUGLAS 369      6
ROBINSON R22              20
ROBINSON R44              38
SCHWEIZER 269              5
SIKORSKY 76                2
dtype: int64
```

The result is much like `value_counts`, but we're allowed run to all kinds of statistical operations on the group, like `sum`, `mean` and `std`. For instance, we could sum the total number of fatalities for each maker by stringing that field on the end followed by the statistical method.

```
accident_list.groupby("latimes_make_and_model")['total_fatalities'].sum()
```

```
latimes_make_and_model
AGUSTA 109                 5
AIRBUS 130                 1
AIRBUS 135                11
AIRBUS 350                81
BELL 206                  61
BELL 407                  35
HUGHES 369                19
MCDONNELL DOUGLAS 369      7
ROBINSON R22              27
```

(continues on next page)

(continued from previous page)

```
ROBINSON R44             71
SCHWEIZER 269             7
SIKORSKY 76              11
Name: total_fatalities, dtype: int64
```

Again our data has come back as an ugly Series. To reformat it as a pretty DataFrame use the `reset_index` method again.

```
accident_list.groupby("latimes_make_and_model").size().reset_index()
```

```
   latimes_make_and_model   0
0              AGUSTA 109   2
1              AIRBUS 130   1
2              AIRBUS 135   4
3              AIRBUS 350  29
4                BELL 206  30
5                BELL 407  13
6              HUGHES 369  13
7   MCDONNELL DOUGLAS 369   6
8            ROBINSON R22  20
9            ROBINSON R44  38
10          SCHWEIZER 269   5
11            SIKORSKY 76   2
```

You can clean up the `0` column name assigned by pandas with the `rename` method.

```
accident_list.groupby("latimes_make_and_model").size().rename("accidents").reset_index()
```

```
   latimes_make_and_model  accidents
0              AGUSTA 109          2
1              AIRBUS 130          1
2              AIRBUS 135          4
3              AIRBUS 350         29
4                BELL 206         30
5                BELL 407         13
6              HUGHES 369         13
7   MCDONNELL DOUGLAS 369          6
8            ROBINSON R22         20
9            ROBINSON R44         38
10          SCHWEIZER 269          5
11            SIKORSKY 76          2
```

Now save that as a variable.

```
accident_counts = accident_list.groupby("latimes_make_and_model").size().rename(
↪"accidents").reset_index()
```

The result is a DataFrame with the accident totals we'll want to merge with the FAA survey data to calculate rates.

Note: You may notice that we've configured `rename` differently than other methods so far. These are what Python calls keyword arguments. They are inputs that are passed to a function or method by explicitly specifying the name of the argument, followed by an equal sign and the value being passed. Here we used `columns` and `inplace`.

Keyword arguments are different from positional arguments, which are passed to a function in the order that they are defined. Keyword arguments can be useful because they make it clear which argument is being passed and also they can be passed in any order, as long as the name of the argument is specified.

Also, keyword arguments are often used to specify default values for a function's parameters, this way if an argument is not passed, the default value will be used. For this reason, they are often used in pandas to override the default behavior and provide customizations beyond the out-of-the-box behavior of a method. Visiting the pandas documentation for any method, here's the page for `rename`, will reveal what options are available.

To see the result, inspect the DataFrame with `head`.

```
accident_counts.head()
```

```
  latimes_make_and_model  accidents
0              AGUSTA 109          2
1              AIRBUS 130          1
2              AIRBUS 135          4
3              AIRBUS 350         29
4                BELL 206         30
```

Now we've got a ranking we can work with.

3.8 Merge

Next we'll cover how to merge two DataFrames together into a combined table. Before we can do that, we need to read in a second file. We'll pull `faa-survey.csv`, which contains annual estimates of how many hours each type of helicopter was in the air. If we merge it with our accident totals, we will be able to calculate an accident rate.

We can read it in the same way as the NTSB accident list, with `read_csv`.

```
import pandas as pd
accident_list = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-
↪notebook/main/docs/src/_static/ntsb-accidents.csv")
accident_counts = accident_list.groupby("latimes_make_and_model").size().reset_index().
↪rename(columns={0: "accidents"})
```

```
survey = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-notebook/
↪main/docs/src/_static/faa-survey.csv")
```

Before you do anything, take a peek at it with with the `head`.

```
survey.head()
```

```
  latimes_make_and_model  total_hours
0              AGUSTA 109       362172
1              AIRBUS 130      1053786
2              AIRBUS 135       884596
3              AIRBUS 350      3883490
4                BELL 206      5501308
```

When joining two tables together, the first step is to look carefully at the columns in each table to find a common column that can be joined. We can do that with the `info` command we learned earlier.

```
accident_counts.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12 entries, 0 to 11
Data columns (total 2 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   latimes_make_and_model  12 non-null     object
 1   accidents               12 non-null     int64
dtypes: int64(1), object(1)
memory usage: 320.0+ bytes
```

```
survey.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12 entries, 0 to 11
Data columns (total 2 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   latimes_make_and_model  12 non-null     object
 1   total_hours             12 non-null     int64
dtypes: int64(1), object(1)
memory usage: 320.0+ bytes
```

You can see that each table contains the `latimes_make_and_model` column. We can therefore join the two files using that column with the pandas `merge` method.

Note: If you are familar with traditional databases, you may recognize that the merge method in pandas is similar to SQL's JOIN statement. If you dig into merge's documentation you will see it has many of the same options.

Merging two DataFrames is as simple as passing both to pandas built-in `merge` method and specifying which field we'd like to use to connect them together. We will save the result into another new variable, which I'm going to call `merged_list`.

```
merged_list = pd.merge(accident_counts, survey, on="latimes_make_and_model")
```

That new DataFrame can be inspected like any other.

```
merged_list.head()
```

```
  latimes_make_and_model  accidents  total_hours
0             AGUSTA 109          2       362172
1             AIRBUS 130          1      1053786
2             AIRBUS 135          4       884596
3             AIRBUS 350         29      3883490
4               BELL 206         30      5501308
```

By looking at the columns you can check how many rows survived the merge, a precaution you should take every time you join two tables.

```
merged_list.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 12 entries, 0 to 11
Data columns (total 3 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   latimes_make_and_model  12 non-null     object
 1   accidents               12 non-null     int64
 2   total_hours             12 non-null     int64
dtypes: int64(2), object(1)
memory usage: 384.0+ bytes
```

You can also verify that the DataFrame has the same number of records as there are values in `accident_totals` column. That's good; If there are no null values, that means that every record in each DataFrame found a match in the other.

3.9 Compute

```
import pandas as pd
accident_list = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-
↪notebook/main/docs/src/_static/ntsb-accidents.csv")
accident_counts = accident_list.groupby("latimes_make_and_model").size().reset_index().
↪rename(columns={0: "accidents"})
survey = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-notebook/
↪main/docs/src/_static/faa-survey.csv")
merged_list = pd.merge(accident_counts, survey, on="latimes_make_and_model")
```

To calculate an accident rate, we'll need to create a new column based on the data in other columns, a process sometimes known as "computing."

In many cases, it's no more complicated than combining two series using a mathematical operator. That's true in this case, where our goal is to divide the total number of accidents in each row into the total hours. That can accomplished with the following:

```
merged_list['accidents'] / merged_list['total_hours']
```

```
0     5.522238e-06
1     9.489593e-07
2     4.521838e-06
3     7.467510e-06
4     5.453249e-06
5     6.150096e-06
6     1.081812e-05
7     1.089544e-05
8     6.732180e-06
9     1.610354e-05
10    4.388560e-06
11    2.184563e-06
dtype: float64
```

The resulting series can be added to your dataframe by assigning it to a new column. You name your column by providing it as a quoted string inside of flat brackets. Let's call this column something brief and clear like `per_hour`.

```
merged_list['per_hour'] = merged_list['accidents'] / merged_list['total_hours']
```

Which, like everything else, you can inspect with the `head` command.

```
merged_list.head()
```

```
  latimes_make_and_model  accidents  total_hours      per_hour
0            AGUSTA 109          2       362172  5.522238e-06
1            AIRBUS 130          1      1053786  9.489593e-07
2            AIRBUS 135          4       884596  4.521838e-06
3            AIRBUS 350         29      3883490  7.467510e-06
4              BELL 206         30      5501308  5.453249e-06
```

You can see that the result is in scientific notation. As is common when calculating per capita statistics, you can multiple all results by a common number to make the numbers more legible. That's as easy as tacking on the multiplication at the end of a computation. Here we'll use 100,000.

```
merged_list['per_100k_hours'] = merged_list['per_hour'] * 100_000
```

3.10 Sort

Another simple but common technique for analyzing data is sorting. This can be useful for ranking the DataFrame to show the first and last members of the table according to a particular column.

```
import pandas as pd
accident_list = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-
↪notebook/main/docs/src/_static/ntsb-accidents.csv")
accident_counts = accident_list.groupby("latimes_make_and_model").size().reset_index().
↪rename(columns={0: "accidents"})
survey = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-notebook/
↪main/docs/src/_static/faa-survey.csv")
merged_list = pd.merge(accident_counts, survey, on="latimes_make_and_model")
merged_list['per_hour'] = merged_list.accidents / merged_list.total_hours
merged_list['per_100k_hours'] = (merged_list.accidents / merged_list.total_hours) * 100_
↪000
```

The `sort_values` method is how pandas does it. It expects that you provide it with the name of the column to sort by in quotes. Try sorting by our computed field.

```
merged_list.sort_values("per_100k_hours")
```

```
   latimes_make_and_model  accidents  total_hours      per_hour  \
1              AIRBUS 130          1      1053786  9.489593e-07
11           SIKORSKY 76           2       915515  2.184563e-06
10         SCHWEIZER 269           5      1139326  4.388560e-06
2              AIRBUS 135          4       884596  4.521838e-06
4                BELL 206         30      5501308  5.453249e-06
0              AGUSTA 109          2       362172  5.522238e-06
5                BELL 407         13      2113788  6.150096e-06
8            ROBINSON R22         20      2970806  6.732180e-06
3              AIRBUS 350         29      3883490  7.467510e-06
```

(continues on next page)

(continued from previous page)

```
6              HUGHES 369          13      1201688  1.081812e-05
7   MCDONNELL DOUGLAS 369           6       550689  1.089544e-05
9            ROBINSON R44          38      2359729  1.610354e-05

    per_100k_hours
1         0.094896
11        0.218456
10        0.438856
2         0.452184
4         0.545325
0         0.552224
5         0.615010
8         0.673218
3         0.746751
6         1.081812
7         1.089544
9         1.610354
```

Note that returns the DataFrame resorted in ascending order from lowest to highest. That is pandas' default way of sorting. You reverse it to show the largest values first by passing in an optional keyword argument called `ascending`. When it is set to `False`, the DataFrame is sorted in descending order.

```
merged_list.sort_values("per_100k_hours", ascending=False)
```

```
   latimes_make_and_model  accidents  total_hours      per_hour  \
9            ROBINSON R44         38      2359729  1.610354e-05
7   MCDONNELL DOUGLAS 369          6       550689  1.089544e-05
6              HUGHES 369         13      1201688  1.081812e-05
3              AIRBUS 350         29      3883490  7.467510e-06
8            ROBINSON R22         20      2970806  6.732180e-06
5                BELL 407         13      2113788  6.150096e-06
0              AGUSTA 109          2       362172  5.522238e-06
4                BELL 206         30      5501308  5.453249e-06
2              AIRBUS 135          4       884596  4.521838e-06
10          SCHWEIZER 269          5      1139326  4.388560e-06
11            SIKORSKY 76          2       915515  2.184563e-06
1              AIRBUS 130          1      1053786  9.489593e-07

    per_100k_hours
9         1.610354
7         1.089544
6         1.081812
3         0.746751
8         0.673218
5         0.615010
0         0.552224
4         0.545325
2         0.452184
10        0.438856
11        0.218456
1         0.094896
```

Congratulations. With that, you've recreated the heart of the analysis published in the Los Angeles Times and covered

most of the basic skills necessary to access and analyze data with pandas. In our next chapter, we will show how you can chart the data with Altair.

3.11 Charts

Python has a number of charting tools that can work hand-in-hand with pandas. While Altair is a relative newbie compared to veterans like matplotlib, it's got great documentation and is easy to configure. Let's take it for a spin.

3.11.1 Make a basic bar chart

Head back to the import cell at the top of your notebook and add Altair. In the tradition of pandas, we'll import it with the alias `alt` to reduce how much we need to type later on.

```
import warnings
warnings.simplefilter('ignore')
import pandas as pd
accident_list = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-
↪notebook/main/docs/src/_static/ntsb-accidents.csv")
accident_counts = accident_list.groupby(["latimes_make", "latimes_make_and_model"]).
↪size().rename("accidents").reset_index()
survey = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-notebook/
↪main/docs/src/_static/faa-survey.csv")
merged_list = pd.merge(accident_counts, survey, on="latimes_make_and_model")
merged_list['per_hour'] = merged_list.accidents / merged_list.total_hours
merged_list['per_100k_hours'] = (merged_list.accidents / merged_list.total_hours) * 100_
↪000
```

```
import altair as alt
```

Once that's run, we can pick up where we last left off at the bottom of the notebook. Let's try to plot our accident rate ranking as a bar chart.

With Altair imported, we can now feed it our DataFrame to start charting. Let's take a look at the basic building block of an Altair chart: the `Chart` object. We'll tell it that we want to create a chart from `merged_list` by passing the dataframe in, like so:

```
alt.Chart(merged_list)
```

```
---------------------------------------------------------------------------
SchemaValidationError                         Traceback (most recent call last)
~/checkouts/readthedocs.org/user_builds/first-python-notebook/envs/stable/lib/python3.7/
↪site-packages/altair/vegalite/v5/api.py in to_dict(self, *args, **kwargs)
   2518             copy.data = core.InlineData(values=[{}])
   2519             return super(Chart, copy).to_dict(*args, **kwargs)
-> 2520         return super().to_dict(*args, **kwargs)
   2521
   2522     def add_params(self, *params) -> Self:

~/checkouts/readthedocs.org/user_builds/first-python-notebook/envs/stable/lib/python3.7/
↪site-packages/altair/vegalite/v5/api.py in to_dict(self, *args, **kwargs)
    848         # but due to how Altair is set up this should hold.
```

(continues on next page)

(continued from previous page)

```
    849         # Too complex to type hint right now
--> 850         dct = super(TopLevelMixin, copy).to_dict(*args, **kwargs)  # type:␣
↪ignore[misc]
    851
    852         # TODO: following entries are added after validation. Should they be␣
↪validated?

~/checkouts/readthedocs.org/user_builds/first-python-notebook/envs/stable/lib/python3.7/
↪site-packages/altair/utils/schemapi.py in to_dict(self, validate, ignore, context)
    812                 # show the less helpful ValidationError instead of
    813                 # the more user friendly SchemaValidationError
--> 814                 raise SchemaValidationError(self, err) from None
    815         return result
    816

SchemaValidationError: '{'data': {'name': 'data-45403fa9afde868a6cd19a10672afb20'}}' is␣
↪an invalid value.

'mark' is a required property
```

```
alt.Chart(...)
```

OK! We got an error, but don't panic. The error says that Altair needs a "mark" — that is to say, it needs to know not only what data we want to visualize, but also *how* to represent that data visually. There are lots of different marks that Altair can use (You can check them all out here). But let's try out the most versatile mark in our visualization toolbox: the bar.

```
alt.Chart(merged_list).mark_bar()
```

```
alt.Chart(...)
```

That's an improvement, but we've got a new error: Altair doesn't know what columns of our dataframe to look at! At a minimum, we also need to define the column to use for the x- and y-axes. We can do that by chaining in the `encode` method.

```
alt.Chart(merged_list).mark_bar().encode(
    x="latimes_make_and_model",
    y="per_100k_hours"
)
```

```
alt.Chart(...)
```

That's more like it!

Here's an idea — maybe we want to do horizontal, not vertical bars. How would you rewrite this chart code to reverse those bars?

```
alt.Chart(merged_list).mark_bar().encode(
    x="per_100k_hours",
    y="latimes_make_and_model"
)
```

```
alt.Chart(...)
```

This chart is an okay start, but it's sorted alphabetically by y-axis value, which is pretty sloppy and hard to visually parse. Let's fix that.

We want to sort the y-axis values by their corresponding x values. We've been using the shorthand syntax to pass in our axis columns so far, but to add more customization to our chart we'll have to switch to the longform way of defining the y axis.

To do that, we'll use a syntax like this: `alt.Y(column_name, arg="value")`. Instead of passing a string to `y`, this lets us pass in a string and then any number of named arguments. There are lots more arguments that you might want to pass in, like ones that will sum or average your data on the fly, or limit the range you want your axis to display. In this case, we'll try out the `sort` option.

```
alt.Chart(merged_list).mark_bar().encode(
    x="per_100k_hours",
    y=alt.Y("latimes_make_and_model", sort="-x")
)
```

```
alt.Chart(...)
```

And we can't have a chart without context. Let's throw in a title for good measure.

```
alt.Chart(merged_list).mark_bar().encode(
    x="per_100k_hours",
    y=alt.Y("latimes_make_and_model", sort="-x")
).properties(
    title="Helicopter accident rates"
)
```

```
alt.Chart(...)
```

Yay, we made a chart!

3.11.2 Other marks

What if we wanted to switch it up and show this data in a slightly different form? For example, in the Los Angeles Times story, the fatal accident rate is shown as a scaled circle.

We can try that out with just a few small tweaks, using Altair's `mark_circle` option. We'll keep the `y` encoding, since we still want to split out our chart by make and model. Instead of an `x` encoding, though, we'll pass in a `size` encoding, which will pin the radius of each circle to that rate calculation. And hey, while we're at it, let's throw in an interactive tooltip.

```
alt.Chart(merged_list).mark_circle().encode(
    size="per_100k_hours",
    y="latimes_make_and_model",
    tooltip="per_100k_hours"
)
```

```
alt.Chart(...)
```

A nice little change from all the bar charts! But once again, this is by default sorted alphabetically by name. Instead, it would be really nice to sort this by rate, as we did with the bar chart. How would we go about that?

```
alt.Chart(merged_list).mark_circle().encode(
    size="per_100k_hours",
    y=alt.Y("latimes_make_and_model", sort='-size'),
    tooltip="per_100k_hours"
)
```

```
alt.Chart(...)
```

3.11.3 Add a `color`

What important facet of the data is this chart *not* showing? There are two Robinson models in the ranking. It might be nice to emphasize them.

We have that `latimes_make` column in our original dataframe, but it got lost when we created our ranking because we didn't include it in our `groupby` command. We can fix that by scrolling back up our notebook and adding it to the command. You will need to replace what's there with a list containing both columns we want to keep.

```
accident_counts = accident_list.groupby(["latimes_make", "latimes_make_and_model"]).
↪size().rename("accidents").reset_index()
```

Rerun all of the cells below to update everything you're working with. Now if you inspect the ranking you should see the `latimes_make` column included.

```
merged_list.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 12 entries, 0 to 11
Data columns (total 6 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   latimes_make            12 non-null     object
 1   latimes_make_and_model  12 non-null     object
 2   accidents               12 non-null     int64
 3   total_hours             12 non-null     int64
 4   per_hour                12 non-null     float64
 5   per_100k_hours          12 non-null     float64
dtypes: float64(2), int64(2), object(2)
memory usage: 672.0+ bytes
```

Let's put that to use with an Altair option that we haven't used yet: `color`.

```
alt.Chart(merged_list).mark_bar().encode(
    x="per_100k_hours",
    y=alt.Y("latimes_make_and_model",sort="-x"),
    color="latimes_make"
).properties(
    title="Helicopter accident rates"
)
```

```
alt.Chart(...)
```

Hey now! That wasn't too hard, was it? But now there's too many colors. We would be better off if we emphasized the Robinson bars, but left the rest of the makers the default color.

We can accomplish that by taking advantage of `alt.condition`, Altair's method for adding logic to the configuration of the chart. In this case, we want to set the chart one color if Robinson is the maker, and another if it isn't. Here's how to do that:

```
alt.Chart(merged_list).mark_bar().encode(
    x="per_100k_hours",
    y=alt.Y("latimes_make_and_model",sort="-x"),
    color=alt.condition(
        alt.datum.latimes_make == "ROBINSON",
        alt.value("orange"),
        alt.value("steelblue")
    )
).properties(
    title="Helicopter accident rates"
)
```

```
alt.Chart(...)
```

3.11.4 `datetime` data

One thing you'll almost certainly find yourself grappling with time and time again is date (and time) fields, so let's talk about how to handle them.

Let's see if we can do that with our original DataFrame, the `accident_list` that contains one record for every helicopter accident. We can remind ourselves what it contains with the `info` command.

```
accident_list.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 163 entries, 0 to 162
Data columns (total 13 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   event_id               163 non-null    object
 1   ntsb_make              163 non-null    object
 2   ntsb_model             163 non-null    object
 3   ntsb_number            163 non-null    object
 4   year                   163 non-null    int64
 5   date                   163 non-null    object
 6   city                   163 non-null    object
 7   state                  162 non-null    object
 8   country                163 non-null    object
 9   total_fatalities       163 non-null    int64
 10  latimes_make           163 non-null    object
 11  latimes_model          163 non-null    object
 12  latimes_make_and_model  163 non-null    object
dtypes: int64(2), object(11)
memory usage: 16.7+ KB
```

When you import a CSV file with `read_csv` it will take a guess at column types — for example, `integer`, `float`, `boolean`, `datetime` or `string` — but it will default to a generic `object` type, which will generally behave like a string, or text, field. You can see the data types that pandas assigned to our accident list on the right hand side of the `info` table.

Take a look above and you'll see that pandas is treating our `date` column as an object. That means we can't chart it using Python's system for working with dates.

But we can fix that. The `to_datetime` method included with `pandas` can handle the conversion. Here's how to reassign the `date` column after making the change.

```
accident_list['date'] = pd.to_datetime(accident_list['date'])
```

This redefines each object in that column as a date. If your dates are in an unusual or ambiguous format, you may have to pass in a specific formatter, but in this case pandas should be able to guess correctly.

Run `info` again and you'll notice a change. The data type for `date` has changed.

```
accident_list.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 163 entries, 0 to 162
Data columns (total 13 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   event_id                163 non-null    object
 1   ntsb_make               163 non-null    object
 2   ntsb_model              163 non-null    object
 3   ntsb_number             163 non-null    object
 4   year                    163 non-null    int64
 5   date                    163 non-null    datetime64[ns]
 6   city                    163 non-null    object
 7   state                   162 non-null    object
 8   country                 163 non-null    object
 9   total_fatalities        163 non-null    int64
 10  latimes_make            163 non-null    object
 11  latimes_model           163 non-null    object
 12  latimes_make_and_model  163 non-null    object
dtypes: datetime64[ns](1), int64(2), object(10)
memory usage: 16.7+ KB
```

Now that we've got that out of the way, let's see if we can chart with it. Let's see if we can count the total fatalities over time.

```
alt.Chart(accident_list).mark_bar().encode(
  x="date",
  y="total_fatalities"
)
```

```
alt.Chart(...)
```

This is great on the x axis, but it's not quite accurate on the y. To make sure this chart is accurate, we'll need to aggregate the y axis in some way.

3.11.5 Aggregate with Altair

We could back out and create a new dataset grouped by date, but Altair actually lets us do some of that grouping on the fly. We want to add everything that happens on the same date, so we'll pop in a `sum` function on our y column.

```
alt.Chart(accident_list).mark_bar().encode(
  x="date",
  y="sum(total_fatalities)"
)
```

```
alt.Chart(...)
```

This is getting there. But sometimes plotting on a day-by-day basis isn't all that useful — especially over a long period of time, like we have here.

Again, we could back out and create a new dataframe grouping by month, but we don't have to — in addition to standard operations (sum, mean, median, etc.), Altair gives us some handy datetime aggregation options. You can find a list of options in the library documentation.

```
alt.Chart(accident_list).mark_bar().encode(
  x="yearmonth(date)",
  y="sum(total_fatalities)",
)
```

```
alt.Chart(...)
```

This is great for showing the pattern of fatalities over time, but it doesn't give us additional information that might be useful. For example, we almost certainly want to investigate the trend for each manufacturer.

We could do that by adding a color encoding, like we did on the last chart. In this case, though, stacking those bars makes it a little hard to focus on amounts individually. What can do instead is to facet, which will create separate charts, one for each helicopter maker.

```
alt.Chart(accident_list).mark_bar().encode(
  x="yearmonth(date)",
  y="sum(total_fatalities)",
  facet="latimes_make"
)
```

```
alt.Chart(...)
```

3.11.6 Polishing your chart

These charts give us plenty of areas where we might want to dig in and ask more questions, but none are polished enough to pop into a news story quite yet. But there *are* lots of additional labeling, formatting and design options that you can dig into in the Altair docs — you can even create Altair themes to specify default color schemes and fonts.

But you may not want to do all that tweaking in Altair, especially if you're just working on a one-off graphic. If you wanted to hand this chart off to a graphics department, all you'd have to do is head to the top right corner of your chart.

See those three dots? Click on that, and you'll see lots of options. Downloading the file as an SVG will let anyone with graphics software like Adobe Illustrator take this file and tweak the design.

To get the raw data out, you'll need to learn one last pandas trick. It's covered in our final chapter.

3.12 Export

Saving the dataframes you've created to your computer requires one final pandas method. It's `to_csv`, an exporting companion to `read_csv`. Append it to any dataframe and provide a filepath. That's all it takes.

```
import pandas as pd
accident_list = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-
↪notebook/main/docs/src/_static/ntsb-accidents.csv")
accident_counts = accident_list.groupby(["latimes_make", "latimes_make_and_model"]).
↪size().reset_index().rename(columns={0: "accidents"})
survey = pd.read_csv("https://raw.githubusercontent.com/palewire/first-python-notebook/
↪main/docs/src/_static/faa-survey.csv")
merged_list = pd.merge(accident_counts, survey, on="latimes_make_and_model")
merged_list['per_hour'] = merged_list.accidents / merged_list.total_hours
merged_list['per_100k_hours'] = (merged_list.accidents / merged_list.total_hours) * 100_
↪000
```

```
merged_list.to_csv("accident-rate-ranking.csv")
```

The file it creates can be imported into other programs for reuse, including the data visualization tools many newsrooms rely on to publish graphics. For instance, the file we've exported above could be used to quickly draft a chart with Datawrapper, like this one:

Note: Interested in learning more about how to publish data online? Check out "First Visual Story," a tutorial that will show you how journalists at America's top news organizations escape rigid content-management systems to publish custom interactive graphics on deadline.

The `to_csv()` method accepts several additional optional arguments. The most important one is the filename input, which is used to specify the path and name of the file that will be created. The `index=False` keyword argument tells pandas to exclude the index column of the DataFrame. You can also specify the separator by passing the `sep` parameter.

```
merged_list.to_csv("accident-rate-ranking.csv", index=False, sep=";")
```

This will create a CSV file without the index with semicolons as the separator between values.

And with that, you've completed "First Python Notebook." If you have any questions or critiques, you can get involved on our GitHub repository, where all of the code that powers this site is available as open source.

3.13 Advanced installation

While there are numerous ways to install and configure Jupyter notebooks, advanced users like to take advantage of Python's power tools to have more control over when and where code is installed on their system.

This guide will demonstrate how to install everything you computer needs to play like the pros.

Sections

3.13.1 A command-line interface

Whether you know about it or not, there should be a way to open a window and directly issue commands to your operating system. Different operating systems give this tool slightly different names, but they all have some form of it.

On Windows this is called the "command prompt." On MacOS it is called the "terminal." Other people will call this the "command line."

On Windows, we recommend you install the Windows Subsystem for Linux and select the Ubuntu distribution from the Windows Store. This will give you access to a generic open-source terminal without all the complications and quirks introduced by Windows. On MacOS, the standard terminal app will work fine.

3.13.2 Python 3.6 or higher

Python is a free and open-source computer programming language. It's one of the most popular in the world and praised by its supporters as clear and easy to read.

That makes it ideal for beginners and is partly why it's been adopted by professionals in many fields, ranging from engineering and web development to journalism and music.

You can check if Python is already installed on your computer by visiting your command line and entering the following:

```
python --version
```

You should see something like this after you hit enter:

```
Python 3.6.10
```

If not, you'll need to install Python on your system.

If you see a number starting with 2, like say ...

```
Python 2.7.12
```

...then you have an outdated version of Python and will need to upgrade to a version starting with a three. You can probably complete the class without doing so, but the maintainers of Python are gradually phasing out version two and officially recommend you upgrade.

Instructions for both new installations and upgrades can be found here.

3.13.3 The `pipenv` environment manager

Our notebook depends on a set of Python tools that we'll need to install before we can run the code. They are the JupyterLab computational notebook, the requests library for downloading webpages and BeautifulSoup, a handy utility for parsing data out of HTML.

By default, Python's third-party packages are all installed in a shared "global" folder somewhere in the depths of your computer. By default, every Python project on your computer draws from this same set of installed programs.

This approach is fine for your first experiments with Python, but it quickly falls apart when you start to get serious about coding.

For instance, say you develop a web application today with Flask version 1.1. What if, a year from now, you want to start a new project and use a newer version of Flask? Your old app is still live and requires occasional patches, but you don't have time to re-write all of your old to make it compatible with the latest version of Flask.

Open-source projects are changing every day and such conflicts are common, especially when you factor in the sub-dependencies of your project's direct dependencies, as well as the sub-dependencies of those sub-dependencies.

Programmers solve this problem by creating a virtual environment for each project that isolates them into discrete, independent containers that do not rely on code in the global environment.

Strictly speaking, working within a virtual environment is not required. At first, it might even feel like a hassle. But in the long run, you will be glad you did it. And you don't have to take my word for it, you can read discussions on StackOverflow and Reddit.

Good thing pipenv can do this for us.

Pipenv and its prerequisites are installed via your computer's command-line interface. You can verify it's there by typing the following into your terminal:

```
pipenv --version
```

If you have it installed, you should see the terminal respond with the version on your machine.

```
pipenv, version 2018.11.26
```

If you get an error, you will need to install it.

If you are on a Mac, Pipenv's maintainers recommend installing via Homebrew:

```
brew install pipenv
```

If you are on Windows and using the Windows Subsystem for Linux, you can install Linuxbrew and use it to install Pipenv.

If neither option makes sense for you, Pipenv's docs recommend a user install via pip:

```
pip install --user pipenv
```

Whatever installation route you choose, you can confirm your success by testing for its version again:

```
pipenv --version
```

If you see that version number now, you know you're okay.

Create a code directory

Now let's create a common folder where all you of your projects will be stored starting with this one. This is also where our virtualenv will be configured.

Depending on your operating system and personal preferences, open up a terminal program. It will start you off in your computer's home directory, just like your file explorer. Enter the `ls` command and press enter to see all of the folders there now.

```
ls
```

Now let's check where we are in our computer's file system. For this we'll use a command called pwd, which stands for present working directory. The output is the full path of your location in the file system, something like `/Users/palewire/`.

```
pwd
```

Use the mkdir command to create a new directory for your code. In the same style as the Desktop, Documents and Downloads folders included by most operating system, we will name this folder Code.

```
mkdir Code
```

To verify that worked, you can open in your file explorer and navigate to your home folder. Now jump into the Code directory, which is the same as double clicking on a folder to enter it in your filesystem navigator.

```
cd Code
```

Create a project directory

Now let's make a folder for your work in this class.

```
mkdir first-python-notebook
```

Then, jump into project directory:

```
cd first-python-notebook
```

This is where you'll store a local copy of all the code and files you create for this project.

It isn't necessary to change directories one level at a time. You can also specify the full path of directory you want to change into. For instance, from your home directory you could running the following to move directly into your project directory.

```
cd Code/first-python-notebook
```

Install your first package

Now let's install a simple Python package to see `pipenv` in action. We'll choose yolk3k, a simple command-line tool that can list all your installed python packages.

We can add it to our project's private virtual environment by typing its name after Pipenv's install command.

```
pipenv install yolk3k
```

When you invoke Pipenv's `install` command, it checks for an existing virtual environment connected to your project's directory. Finding none, it creates one, then installs your packages into it.

As a result, two files are added to your project directory: `Pipfile` and `Pipfile.lock`. Open these files in a text editor and you'll see how they describe your project's Python requirements.

In the `Pipfile`, you'll see the name and exact version of any package we directed Pipenv to install. We didn't specify an exact version, so you'll see:

```
[packages]
yolk3k = "*"
```

`Pipfile.lock` has a more complicated, nested structure that specifies the exact version of your project's direct dependencies along with all their sub-dependencies.

Now that yolk is installed, we can execute it inside our environment using the `pipenv run` command. Let's use it to see yolk3k's method for listing all of our currently installed tools.

```
pipenv run yolk -l
```

You should see the computer spit out everything you have installed. You'll notice that yolk3k is on the list.

3.13.4 Python packages

Next we will install the extra Python packages used during the tutorial.

We will return to pipenv and use it to install JupyterLab, the web-based interactive development environment for Jupyter notebooks, code and data.

```
pipenv install jupyterlab
```

We'll install pandas the same way:

```
pipenv install pandas
```

Install altair too.

```
pipenv install altair
```

Note: You can install more than one package at once. For instance, all three of the packages above could be added like so:

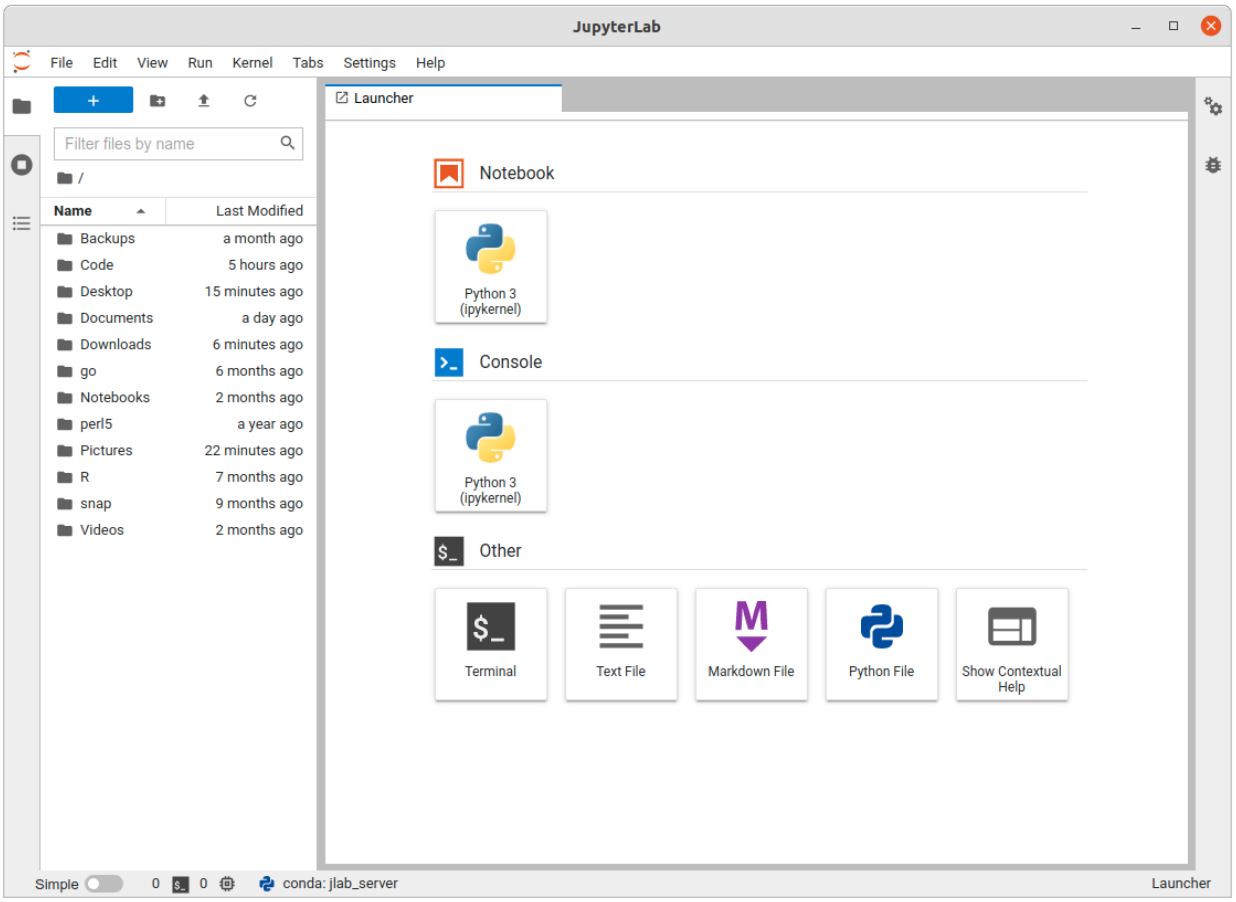
```
pipenv install jupyterlab pandas altair
```

3.13.5 Your first notebook

Now we can use pipenv's run command to start JupyterLab from your terminal.

```
pipenv run jupyter lab
```

That will open up a new tab in your default web browser that looks something like this:



Click the "Python 3" button in the middle panel and create a new Python 3 notebook. You should now be able to pick up in *chapter two* and start work from there.

3.14 About this class

This course was first developed by Ben Welsh for an October 2016 "watchdog workshop" organized by Investigative Reporters and Editors at San Diego State University's school of journalism.

Since then it has grown into a six-hour, hands-on training that has been taught as a massive, open online course at the University of Texas at Austin, become part of Stanford's curriculum and been offered at an unbroken string of annual conferences organized by the National Institute for Computer-Assisted Reporting. Several classes have been streamed live, with more than one available as a recording.

The class has frequently been taught with and by others, including James Gordon, Andrea Suozzo, Cheryl Phillips, Iris Lee, Gabrielle LaMarr LeMee, Melissa Lewis, Aaron Williams, Derek Willis, Joe Germuska, Kae Petrin, Eric Sagara, Serdar Tumgoren, Simon Willison, David Eads and Amy Schmitz Weiss.